Elements of Programming Languages
Tutorial 4: Subtyping and polymorphism
Solution notes

1. Subtyping and type bounds

(a) 
\[ \text{Sub}1 <: \text{Super} \quad \text{Sub}2 <: \text{Super} \]

(b) i. \( \text{Sub}1 \times \text{Sub}2 <: \text{Super} \times \text{Super} \) This holds:
\[
\frac{\text{Sub}1 <: \text{Super} \quad \text{Sub}2 <: \text{Super}}{\text{Sub}1 \times \text{Sub}2 <: \text{Super} \times \text{Super}}
\]

ii. \( \text{Sub}1 \rightarrow \text{Sub}2 <: \text{Super} \rightarrow \text{Super} \) This does not hold since \( \text{Super} <: \text{Sub}1 \) doesn’t.
\[
\frac{\text{Super} <: \text{Sub}1 \quad \text{Sub}2 <: \text{Super}}{\text{Sub}1 \rightarrow \text{Sub}2 <: \text{Super} \rightarrow \text{Super}}
\]

iii. \( \text{Super} \rightarrow \text{Super} <: \text{Sub}1 \rightarrow \text{Sub}2 \) This does not hold since \( \text{Super} <: \text{Sub}2 \) doesn’t.
\[
\frac{\text{Sub}1 <: \text{Super} \quad \text{Sub}2 <: \text{Super}}{\text{Super} \rightarrow \text{Super} <: \text{Sub}1 \rightarrow \text{Sub}2}
\]

iv. \( \text{Super} \rightarrow \text{Sub}1 <: \text{Sub}2 \rightarrow \text{Super} \) This holds:
\[
\frac{\text{Sub}1 <: \text{Super} \quad \text{Sub}2 <: \text{Super}}{\text{Super} \rightarrow \text{Sub}1 <: \text{Sub}2 \rightarrow \text{Super}}
\]

v. \((\ast)\) \( (\text{Sub}1 \rightarrow \text{Sub}1) \rightarrow \text{Sub}2 <: (\text{Super} \rightarrow \text{Sub}1) \rightarrow \text{Super} \) This holds:
\[
\frac{\text{Sub}1 <: \text{Super} \quad \text{Sub}1 <: \text{Sub}1}{(\text{Super} \rightarrow \text{Sub}1) \rightarrow \text{Sub}1 \rightarrow \text{Sub}1 \rightarrow \text{Sub}2 <: (\text{Super} \rightarrow \text{Sub}1) \rightarrow \text{Super}}
\]

(c) If we call \( f1 \) on \( \text{Sub}2(\text{true}) \) then the result has type \text{Super}. We can’t access the \( b \) field because of a type mismatch.

(d) This typechecks, because in either case we return \( x \) which has type \( A \). If we apply it to a value of type \( \text{Sub}1 \) or \( \text{Sub}2 \) we get the same value back. If we apply it to \( 42 : \text{Int} \) then we get a match error.

(e) This typechecks, because as for \( f2 \) we return \( x : A \) in either case. However, now if we apply to \( \text{Sub}1 \) or \( \text{Sub}2 \) we get the same value back, while if we apply to something of an unrelated type we get a type error. This seems to solve the problem.

2. Subtyping and Contravariance

(a) \( f \) could call its function argument on any \text{Shape}, e.g. either \text{Circle} or \text{Rectangle}. Thus, calling \( f \) on a function of type \text{Rectangle} -> \text{Int} is not allowed, because \text{Rectangle} -> \text{Int} is not a subtype of \text{Shape} -> \text{Int}. If this call was executed, then \( f \) could call its argument on a \text{Circle}, which would not match the expected \text{Rectangle} argument type.
(b) \( g \) can only call its function argument on a \( \text{Circle} \). Thus, calling \( g \) on a function of type \( \text{Shape} \rightarrow \text{Int} \) is allowed, because \( \text{Shape} \rightarrow \text{Int} \) is a subtype of \( \text{Circle} \rightarrow \text{Int} \). If we execute this call, then whatever \( g \) does with its function argument will be fine, since the expected type of the function argument is \( \text{Shape} \), so it can handle any particular type of shape such as \( \text{Circle} \).

3. Type parameters

(a)

```scala
abstract class Tree[A]
case class Leaf[A](a: A) extends Tree[A]
case class Node[A](t1: Tree[A], t2: Tree[A]) extends Tree[A]
```

(b)

```scala
def sum(t: Tree[Int]): Int = t match {
  case Leaf(a) => a
  case Node(t1, t2) => sum(t1) + sum(t2)
}
```

(c)

```scala
def map[A,B](t: Tree[A])(f: A => B): Tree[B] = t match {
  case Leaf(a) => Leaf(f(a))
  case Node(t1, t2) => Node(map(t1)(f), map(t2)(f))
}
```

(d)

```scala
def flatten[A](t: Tree[Tree[A]]): Tree[A] = t match {
  case Leaf(u) => u
  case Node(t1, t2) => Node(flatten(t1), flatten(t2))
}
```

(e)

```scala
def flatMap(t: Tree[A])(f: A => Tree[B]) = flatten(map(t)(f))
```